BTA Termination Using CFL-Reachability

Manuvir Das Thomas Reps

University of Wisconsin-Madison

Abstract

In this paper, we develop a BTA algorithm that ensures termination of off-line partial evaluation for first-order functional programs, extending the work of Holst and of Glenstrup and Jones. Holst's characterization of in-situ-decreasing behaviour does not account for parameters that do not control the recursion of their functions. We extend Holst's framework to handle this phenomenon by defining the "influential" property for parameters. Glenstrup and Jones's algorithm for identifying in-situ-decreasing parameters, which relies on the size markings ($\uparrow$, $\downarrow$, $=$) on the edges of a dependence graph, says that a path must be free of $\uparrow$ edges and must contain at least one $\downarrow$ edge to be size-decreasing. We extend their language of size-decreasing paths: A size-decreasing path may contain $\uparrow$ edges provided that every $\uparrow$ edge is balanced by the appropriate kind of $\downarrow$ edge. We modify the size markings on the graph edges and use Context-Free-Language Reachability, a generalized form of graph reachability, to identify such complex size-decreasing paths.

Keywords: partial evaluation, binding-time analysis, termination, context-free-language reachability

Manuvir Das
Computer Sciences Department
University of Wisconsin-Madison
1210 W. Dayton St.
Madison, WI 53706, USA
manuvir@cs.wisc.edu

Thomas Reps
Computer Sciences Department
University of Wisconsin-Madison
1210 W. Dayton St.
Madison, WI 53706, USA
reps@cs.wisc.edu
Tel: 1 (608) 262-2091
Fax: 1 (608) 262-9777

1 Introduction

This paper concerns the definition of binding-time analyses that ensure termination of off-line partial evaluation of functional programs. The role of a binding-time analysis (BTA) is to annonate all statements in a subject program as either "static" or "dynamic". A typical basis for binding-time analysis is the notion of congruence [5]: A BTA is congruent if no statements annotated "static" by the BTA depend on statements annotated "dynamic". The annotated program is passed to a specializer, which executes the statements marked static and produces residual code for the statements marked dynamic. A known problem with off-line partial evaluators that use congruence-based binding-time analysis is that they may fall into an infinite loop or an infinite recursion because of "static-infinite computations" in a subject program [5, 13] (e.g. infinite loops that are completely static.) In this paper, we develop a BTA algorithm that marks every "static-infinite computation" as "dynamic", extending the work of Holst [4] and of Glenstrup and Jones [3].

As pointed out by Jones in [6], partial evaluation involves a basic tradeoff between "totality" and "computational completeness". If a partial evaluator is computationally complete — in the sense that it executes every static computation in its subject programs — then on a program that contains static-infinite computations the partial evaluator must diverge without producing a residual program. On the other hand, a partial evaluator can be total by attempting to execute only static computations that are guaranteed to terminate, but it will lack computational completeness as a result. In this paper we focus on partial evaluators that aim towards totality.

In the context of functional programs without looping constructs and infinite data structures, non-terminating behaviour results from infinite recursion. Holst has shown that in programs that manipulate list data it is possible to identify functions that are limited to finite recursion [4]. He identifies parameters that are "in-situ decreasing": An in-situ decreasing parameter of a function f strictly decreases in size on every (recursive) chain of calls from f to f. A function that contains an in-situ decreasing parameter can only call itself a finite number of times before this parameter takes on the value *null*.[1]

Glenstrup and Jones have defined a second algorithm that identifies in-situ decreasing parameters [3]. They define a structure, called the parameter dependency graph (PDG), whose edges denote data dependences between function parameters. Edges are labelled to indicate their size-changing effects, as in Example 1 below. In this framework, a "size-decreasing path" is a path free of $\uparrow$ edges but containing at least one $\downarrow$ edge. An in-situ decreasing parameter is one for which every path in the PDG from the parameter to itself is size decreasing. Such parameters can be identified by solving a simple reachability problem on the PDG: A parameter is in-situ decreasing if its vertex in the PDG is reachable from itself only via paths that are size-decreasing. The presence of in-situ decreasing parameters is used to classify some static statements as "bounded-static-varying" (BSV); BSV statements can always be executed with terminating behaviour. All statements not marked as BSV (including some previously marked "static") are reclassified as dynamic, and the modified annotations are passed to the specialization phase.

A limitation of Glenstrup and Jones's method for identifying in-situ decreasing parameters is the requirement that size-decreasing paths must be free of $\uparrow$ edges. In general, size-decreasing paths may include $\uparrow$ edges that are "matched" by $\downarrow$ edges, as is the case for a path from parameter *vals* of function *eval* to itself in Example 1 below.

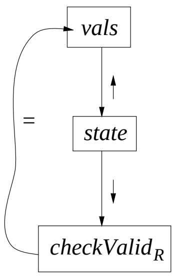
Example 1 In program $\mathbf{P_1}$ below, *eval* is an infix expression evaluator that take a list of operators (*ops*), a list of values (*vals*), the current value of the expression (*tot*), and an error token (*err*) that it returns if an invalid operator is encountered. Function *checkValid* sets the list of remaining values to *null* if an invalid

[1] As discussed later in this section, an in-situ decreasing parameter may not control the recursion of the function, in which case operations such as car(null) may be executed, resulting in an error or in infinite unrolling.

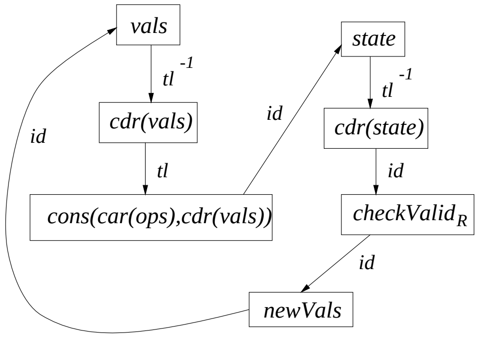
operator is detected, while *accum* updates the expression value at each step.

```
P1 :  eval(ops,vals,tot,err)
        if ops = null and vals = null
          tot
        else if vals = null
          err
        else
          eval(cdr(ops),newVals,newTot,err)
            where newVals = checkValid(cons(car(ops),cdr(vals)))
                  newTot = accum(car(ops),car(vals),tot)

      checkValid(state)
        if invalid(car(state))
          null
        else
          cdr(state)

      accum(op,val,totVal)
        if op = '+
          totVal + val
        else if op = '-
          totVal - val
```

On each successive call to *eval*, the operations *cdr*, *cons*, and *cdr* are applied to parameter *vals*. The net effect is that *cdr(vals)* (or *null*) is passed to the next call on *eval*. The graph shown in (a) below is a snippet of the PDG for P_1 as defined by Glenstrup and Jones, while the graph in (b) is a snippet of the modified PDG used in our approach. Each graph represents the same cycle from vertex *vals* to itself that is present in each PDG.



(a)



(b)

The path from *vals* to itself in (a) contains an ↑ edge, and is therefore not a size-decreasing path under Glenstrup and Jones's approach. For the corresponding path in (b), the label *tl* on the edge from *cdr(vals)* to *cons(car(ops),cdr(vals))* indicates that the value *cons(car(ops),cdr(vals))* has the value *cdr(vals)* as its tail, while the label tl^{-1} from *state* to *cdr(state)* indicates that the value *cdr(state)* is obtained by extracting the tail of *state*. This allows our technique to determine that the path is size-decreasing: Because the *cons* operation on *cdr(vals)* is balanced by the *cdr* operation on *state*, the net effect is that just a single *cdr* is applied to *vals*. □

To handle such cases, we use CFL-reachability [9], a generalized form of graph reachability. A CFL-reachability problem is one in which a path is considered to connect two vertices in a graph only if the concatenation of the labels on the edges of the path is a word in a certain context-free language. Thus the path from parameter *vals* to itself in example (b) above has the concatenated label $tl^{-1}.tl.id.tl^{-1}.id.id.id$, which is in the language *insitu_decr_path* defined later in the paper (*insitu_decr_path* is the language that defines the notion of size-decreasing paths used in this paper.)

Andersen and Holst have also addressed the problem of identifying in-situ-decreasing parameters more precisely, using a different formalism than the one we use. A comparison of our work with their results is given in Section 8.

Another drawback of Glenstrup and Jones's approach is that an in-situ decreasing parameter may not control the recursion of its associated function. If specialization continues after the parameter has taken on the value *null*, the specializer may attempt to perform operations such as cdr*(null)* or car*(null)*. If the specializer detects such errors, the specialization phase will terminate with an error, while infinite recursion may result if such errors are ignored. This effect is illustrated by the following example:

Example 2 In program $\mathbf{P_1}$ from Example 1, parameter *ops* strictly decreases in size each time *eval* is invoked in a recursive manner. Therefore *ops* is in-situ-decreasing. However, *eval* can call itself recursively if *ops* takes on the value *null* (if $ops = null$ and $vals \neq null$.) During specialization, if *eval* is unfolded after *ops* takes on the value *null*, the specializer will try to evaluate the expression cdr*(null)*. □

In our terminology, parameter *ops* is not "influential" because it does not influence the cessation of recursion for function *eval*. Thus, a parameter must be both "in-situ decreasing" and "influential" for its associated function to be unrolled safely during the specialization phase. We refer to parameters that are both in-situ decreasing and influential as "controlling". We provide a semantic characterization of such parameters. This allows us to provide a firm semantic foundation for our BTA termination algorithm, which we define as a CFL-reachability operation (on an augmented version of the PDG) that approximates the semantic definitions.

The contributions of the paper can be summarized as follows:

- We give a semantic characterization of when a parameter is *controlling*.
  - A parameter is *in-situ-decreasing* iff it strictly decreases in size on every recursive call from its associated function to itself.
  - A parameter is *influential* iff a null value for the parameter guarantees no further recursive calls to its associated function.
  - A parameter is *controlling* iff it is both in-situ-decreasing and influential.
- We provide an algorithm that identifies a subset of all controlling parameters.
  - The algorithm uses CFL-reachability to identify more in-situ-decreasing parameters the algorithm of Glenstrup and Jones.
- We give a semantic characterization of when a parameter is *quasi-BSV* (quasi-bounded-static-varying.)
  - A parameter is *quasi-BSV* if it has a sibling parameter that is controlling. A quasi-BSV parameter will take values limited to finite changes on its entry values. Hence if the set of entry values for the parameter is finite, the parameter itself will be BSV.
- We provide an algorithm that identifies a subset of all BSV parameters.
  - The reachability algorithm uses information about quasi-BSV parameters to propagate BSV information from the program's input parameters.

In addition to a number of other differences between our work and that of Glenstrup and Jones, there is a significant difference in the structure of the two algorithms: In their algorithm, parameters may be identified as in-situ-decreasing because other parameters are BSV, and vice versa. This mutual dependence means that the two phases of their algorithm must be applied iteratively to identify the maximum number of BSV parameters. In our algorithm, in-situ-decreasing parameters are identified once and for all, and then this information is used to identify BSV parameters. There is no need to iterate the two phases.

The rest of the paper is organized as follows: In Section 2, we present an overview of the subject functional language and its semantics. In Section 3, we use this semantics to define the in-situ-decreasing, influential, controlling, quasi-BSV, and initially-bounded properties. In Section 4, we present the augmented data dependence graph (ADDG), an extended form of the parameter dependency graph. In Section 5, we define several CFL-path languages on the ADDG. In Section 6, we relate the in-situ decreasing property with the presence or absence of certain CFL-paths in the ADDG. In Section 7, we present an algorithm that uses the presence or absence of certain CFL-paths in the ADDG to identify a subset of all BSV parameters in a program. In Section 8, we discuss related work.

2 A simple functional language and its semantics

In this section we present a simple, first-order call-by-value functional language, `F`, and the semantics of programs in `F`. The language and its description is taken directly from Glenstrup and Jones's work in [3]; we use the same language so that our results can be compared with previous work, while we reproduce the language description here for completeness. The language `F` is defined by the grammar below:

$$\begin{array}{llcl} p & : \textit{Program} & ::= & f1(x_1,\ldots,x_{m_1}) \;=\; e_1 \quad \ldots \quad fn(z_1,\ldots,z_{m_n}) \;=\; e_n \\ e & : \textit{Expression} & ::= & se \mid \textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 \mid e_1 \textbf{ where } x \;=\; e_2 \mid fi(se_1,\ldots,se_{m_i}) \\ se & : \textit{SimpleExpression} & ::= & be \mid \textit{basefcn}(be_1,\ldots,be_k) \\ be & : \textit{BasicExpression} & ::= & x \mid \textit{constant} \end{array}$$

Base functions consist of *cons*, *car* and *cdr*, all of which have the usual meanings. When dealing with the semantics of programs in `F` we treat the *tail recursive* subset of `F`. However our algorithms apply to all programs in `F`.

2.1 Concrete semantics of programs in `F`

A *state* is a pair $(f,\vec{v})$ with f defined in p, where $\vec{v}$ is a shorthand for $(v_1,\ldots,v_n)$, $n = \textit{arity}(f)$, and $\vec{v}_i$ is the ith component of a tuple. Given two values v_1 and v_2 in a value set V of S-expressions, $v_1 < v_2$ iff v_1 is a substructure of v_2. Any vector $\vec{v}^1$ of input values for the parameters of the main function ($f1$ or *main*) can be split into the pair $[\vec{v}^1_s,\vec{v}^1_d]$ where $\vec{v}^1_s \in \vec{V}_s$ is a vector of values for the known parameters of *main* and $\vec{v}^1_d \in \vec{V}_d$ is a vector of values for the unknown parameters of *main*.

Given a value set V, *call-free evaluation* of expression e is defined as:

$$[\![e]\!]_0\vec{v} = \begin{cases} w, & \text{if } e\text{'s value } w \text{ on } \vec{v} \text{ is computable without function calls.} \\ (g,\vec{w}), & \text{if } e\text{'s value on } \vec{v} \text{ is } g\text{'s value on } \vec{w}. \end{cases}$$

A single step *state transition*, written $(f,\vec{v}) \to (g,\vec{w})$, occurs if p contains $f\vec{x} = e$, and $[\![e]\!]_0\vec{v} = (g,\vec{w})$. Total evaluation of an expression is then defined as:

$$[\![e]\!]\vec{v} = \begin{cases} v, & \text{if } e\text{'s value } v \text{ on } \vec{v} \text{ is computable without function calls.} \\ [\![\textit{body}(g)]\!]\vec{w}, & \text{if } [\![e]\!]_0\vec{v} = (g,\vec{w}) \end{cases}$$

A multi-step *transition sequence* $\tau = [(f^1,\vec{v}^1),\ldots,(f^k,\vec{v}^k)]$ is obtained by composing several state transitions $(f^1,\vec{v}^1) \to (f^2,\vec{v}^2)$, $(f^2,\vec{v}^2) \to (f^3,\vec{v}^3)$, etc. Every recursive call from a function f to itself is associated with a transition sequence of the form $[(f,\vec{v}^1),\ldots,(f,\vec{v}^k)]$ where $\vec{v}^1$ is the vector of the values of f's parameters at the caller and $\vec{v}^k$ is the vector of the values of f's parameters at the callee.

3 Semantically controlling behaviour

As mentioned in the introduction, the fact that a function has an in-situ-decreasing parameter does not guarantee that a function can be unrolled without error. In this section we define the in-situ-decreasing property for function parameters, as well as the "influential" property; these two properties can be combined to identify parameters that can limit a function to finite recursion.

A parameter is in-situ-decreasing if it strictly decreases in size each time its associated function is called recursively. This is characterized formally in Definition 1 below:

Definition 1 A parameter f_j of function f in p is *in-situ-decreasing* iff for every transition sequence $\tau = [(f^1,\vec{v}^1),\ldots,(f^k,\vec{v}^k)]$ where $f^1 = f^k = f$, $\vec{v}^k_j < \vec{v}^1_j$. □

A parameter such as f_j in the definition above will take a *null* value after some number n of recursive calls to f, where n is determined by the size of f_j at the first call to f. For instance, parameter *ops* of function *eval* from Example 1 is in-situ-decreasing as it must decrease in size each time *eval* calls itself.

As can be seen from Example 2 however, function *eval* will be unfolded after parameter *ops* has taken on the value *null*, resulting in an error when cdr*(null)* is performed. This is because parameter *ops* does not influence the recursive behaviour of *eval*. This idea is formalised through the property defined below:

Definition 2 A parameter f_j of function f in p is *influential* iff for every transition sequence $\tau = [(f^1, \vec{v}^1), \ldots, (f^k, \vec{v}^k)]$ where $k > 1$ and $f^1 = f^k = f$, $\vec{v}^1_j \neq null$. □

Definition 2 says that there is no transition sequence from f to f in which parameter f_j has the value *null*. In other words, a null value for f_j guarantees that there will be no further recursive call to f in the execution of p.

Example 3 In program $\mathbf{P_1}$ from Example 1, function *eval* cannot call itself if *vals* takes on the value *null*. Hence parameter *vals* is influential. □

Together, the in-situ-decreasing and influential properties limit the number of levels of recursion that a function can go through: the in-situ-decreasing property provides the assurance that a parameter will take on the value *null* at some point in the recursion of the function, while the influential property guarantees that when the parameter takes on the value *null*, recursion of the function will stop. Definition 3 below formalises this notion.

Definition 3 A parameter f_j of function f in p is *controlling* iff f_j is in-situ-decreasing and f_j is influential. □

A parameter that is controlling may cause BSV behaviour in itself and in its sibling parameters. More precisely, if parameter f_j above is controlling, function f in the definition above must go through only a finite number of recursion steps. Thus for all other parameters f_i of f, the values taken by f_i are limited to finite changes of the entry values of f_i. Such parameters are characterized below.

Definition 4 A parameter f_i of function f in p is *quasi-BSV* iff there is a parameter f_j of function f such that f_j is controlling. □

The values of a quasi-BSV parameter are limited to finite changes on its entry values. Hence if the set of entry values for the parameter is finite, the parameter itself will be BSV. We say that a parameter is "initially-bounded" if for any static input the set of all entry values for the parameter if finite. This concept is formalized below:

Definition 5 A parameter f_j of function f in p is *initially bounded* iff for every static input $\vec{v}^1_s \in \vec{V}_s$, $\{ \vec{v}^k_j \mid \exists\ \tau = [(f^1, [\vec{v}^1_s, \vec{v}^1_d]), \ldots, (f^k, \vec{v}^k)]$ s.t. $\vec{v}^1_d \in \vec{V}_d$, $f^1 = main$, $f^k = f$ and $\forall i \in (2, k-1)\ f^i \neq f\ \}$ is a finite set. □

A parameter that is quasi-BSV and initially-bounded will take on only finitely many different values, and is therefore BSV.

Theorem 1 A parameter f_i of function f in p is BSV if f_i is quasi-BSV and f_i is initially bounded. □

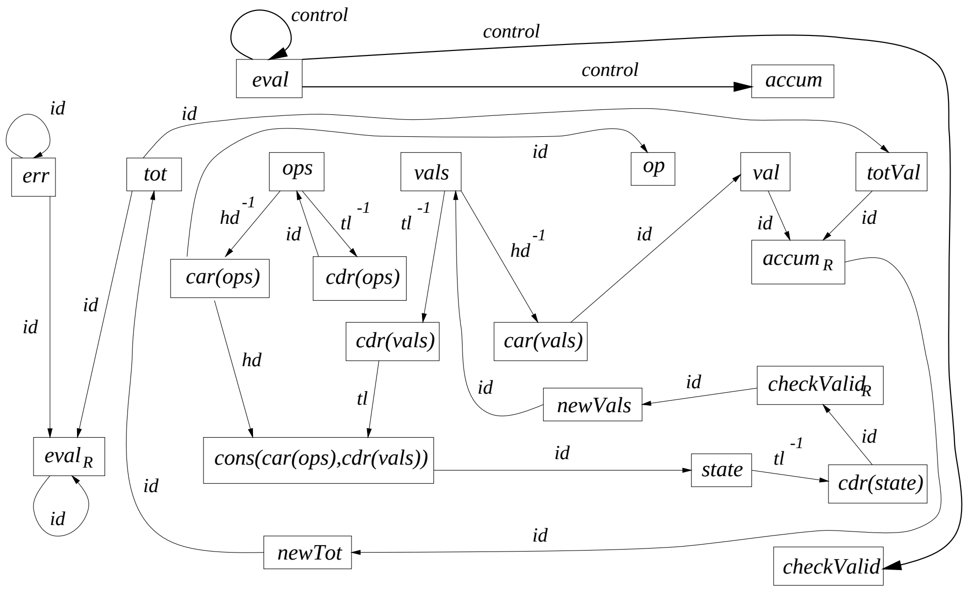
4 The augmented data dependence graph

In this section we present the augmented data dependence graph (ADDG), an extension of Glenstrup and Jones' parameter dependency graph, in which data and control flow in a program is captured through edges in the graph.

The *augmented data dependence graph* for a program p is a directed graph $G(p) = (V, E)$, where V is a set of vertices and E is a set of edges. $V(G)$ includes a header vertex and return value vertex for each function in p, a constant vertex for every function in p whose body includes any constants (other than *null*), a vertex for every intermediate expression in the body of any function in p, a vertex for each variable in p, and a vertex for every parameter of p. $E(G)$ includes flow edges and control edges. Control edges in $E(G)$ are identical to call edges in the call grafsph of p (a control edge exists from f to g iff the expression body of f contains a call on g.) Every control edge has the label *control*. Flow edges in $E(G)$ represent standard data-dependence relationships between vertices (a flow edge exists from u to v if the expression at v is defined in terms of the expression at u.) Every flow edge in G has a label from the set $\{id, hd, tl, hd^{-1}, tl^{-1}\}$, where the label on the edge is determined as follows: edge e from vertex u to vertex v has the label l where

$l = id$	if v is a parameter vertex or if v is a return-value vertex
$l = hd$	if $v = cons(u, w)$
$l = tl$	if $v = cons(w, u)$
$l = hd^{-1}$	if $v = car(u)$
$l = tl^{-1}$	if $v = cdr(u)$

Example 4 The ADDG for program P_1 from Example 1 is shown below.



There are three paths that have vertex *vals* as their target vertex: The path from vertex *vals* with concatenated label $tl^{-1}.tl.id.tl^{-1}.id.id.id$, the path from vertex *ops* with concatenated label $hd^{-1}.hd.id.tl^{-1}.id.id.id$, and the path from vertex *env* with concatenated label $tl^{-1}.id.hd^{-1}.hd.id.tl^{-1}.id.id.id$. □

The number of vertices in the ADDG G of program p is bounded by $\mathcal{O}(P + Var + Ops)$, where P is the number of parameters in p, Var is the number of *where* variables in p, and Ops is the number of cons/car/cdr operations in p. The number of edges in G is bounded by $\mathcal{O}((P + Var)^2 + Ops)$.

5 Paths in the ADDG

In Section 3 we gave a semantic characterization of "in-situ-decreasing parameters". Our purpose in this section is to characterize a subset of the in-situ-decreasing parameters in a program using the augmented data dependence graph defined in the previous section. More accurately, we use the presence or absence of certain kinds of paths in the ADDG to determine whether given parameter vertices in the ADDG satisfy the in-situ-decreasing property. We do this by solving several path problems in which a path is considered to connect two vertices only if the concatenation of the labels on the edges of the path is a word in a certain context-free language.

Definition 6 (Context-Free-Language Reachability; CFL-Reachability) Let L be a context-free language over alphabet Σ, and let G be a graph whose edges are labelled with members of Σ. Each path in G defines a word over Σ, namely, the word obtained by concatenating, in order, the labels on the edges on the path. A path in G is an *L-path* if its word is a member of L. The *all-pairs L-path problem* is to determine all pairs of vertices $v_1, v_2 \in V(G)$ such that there exists an *L-path* in G from v_1 to v_2. The *source-target L-path problem* is to determine whether there exists an *L-path* in G from a given source v_1 to a given target v_2. □

Ordinary reachability (transitive closure) is a degenerate case of CFL-reachability: Let all edges of a graph be labelled with the letter e; transitive closure is the all-pairs e^*-path problem. More general instances of CFL-reachability are useful for focusing on certain paths of interest. By choosing an appropriate language L, we are able to enforce certain types of restrictions on when two vertices are considered to be "connected" (beyond just "connected by a sequence of edges, as one has with ordinary reachability).

CFL-reachability problems can be solved using a dynamic-programming algorithm. (The algorithm can be thought of as a generalization of the CYK algorithm for context-free recognition [7, 16].) There is a general result that all CFL-reachability problems can be solved in time cubic in the number of vertices in the graph [15]. Because the number of vertices in the ADDG for a program p is bounded by $\mathcal{O}(P+Var+Ops)$, where P is the number of parameters in p, Var is the number of *where* variables in p, and Ops is the number of cons/car/cdr operations in p, the problem of identifying whether an L-path exists from every vertex in the ADDG to every other vertex (the all-pairs L-path problem) can be solved in time $\mathcal{O}((P+Var+Ops)^3)$.

Every flow edge from a vertex u in the ADDG to a vertex v in the ADDG has a label from the alphabet $\{id, hd, tl, hd^{-1}, tl^{-1}\}$, indicating the relationship between the values at u and v along a certain execution path. Similarly, the concatenated label on a path from vertex u to vertex v indicates the relationship between the structures of the values at u and v along a certain execution path. We define several contex-free languages such that all paths whose labels are in a given language L relate the values at their source and target vertices in some particular manner (For instance, all L_1-paths as defined below connect vertices whose values are equal along some execution path.) We can then solve the all-pairs L-path problem to determine whether parameters u and v are related in the given manner along some execution path.

We define the following context-free languages:

- The language L_1 represents paths in which each hd(resp. tl) is balanced by a $hd^{-1}(tl^{-1})$; these paths correspond to values transmitted along execution paths in which each cons operation (which gives rise to a hd or tl label on an edge in the path) is eventually "taken apart" by a car(hd^{-1}) or cdr(tl^{-1}) operation:

$$
\begin{aligned}
L_1\!:\ \ & \mathit{eq\_path} \to \mathbf{hd}\ \ \mathit{eq\_path}\ \ \mathbf{hd^{-1}}\ \ \mathit{eq\_path}\\
& \mathit{eq\_path} \to \mathbf{tl}\ \ \mathit{eq\_path}\ \ \mathbf{tl^{-1}}\ \ \mathit{eq\_path}\\
& \mathit{eq\_path} \to \mathbf{id}\ \ \mathit{eq\_path}\\
& \mathit{eq\_path} \to \varepsilon
\end{aligned}
$$

- An L_2-path is a path that has one or more $hd^{-1}(tl^{-1})$ labels that is not balanced by any $hd(tl)$ label. Such paths are "decreasing" in the sense that they correspond to execution paths in which the value at the target vertex of the path is a proper substructure of the value at the source vertex:

$$
\begin{aligned}
L_2\text{:}\quad & \mathit{decr\_path} \rightarrow \mathit{eq\_path}\ \ \mathbf{hd^{-1}}\ \ \mathit{decr\_path} \\
& \mathit{decr\_path} \rightarrow \mathit{eq\_path}\ \ \mathbf{tl^{-1}}\ \ \mathit{decr\_path} \\
& \mathit{decr\_path} \rightarrow \mathit{eq\_path}\ \ \mathbf{hd^{-1}}\ \ \mathit{eq\_path} \\
& \mathit{decr\_path} \rightarrow \mathit{eq\_path}\ \ \mathbf{tl^{-1}}\ \ \mathit{eq\_path}
\end{aligned}
$$

- Similarly, L_3-paths have zero or more $hd^{-1}(tl^{-1})$ labels that are not balanced by any $hd(tl)$ label. Such paths are "equal or decreasing":

$$
\begin{aligned}
L_3\text{:}\quad & \mathit{eq\_or\_decr\_path} \rightarrow \mathit{eq\_path}\ \ \mathbf{hd^{-1}}\ \ \mathit{eq\_or\_decr\_path} \\
& \mathit{eq\_or\_decr\_path} \rightarrow \mathit{eq\_path}\ \ \mathbf{tl^{-1}}\ \ \mathit{eq\_or\_decr\_path} \\
& \mathit{eq\_or\_decr\_path} \rightarrow \mathit{eq\_path}
\end{aligned}
$$

- The languages of L_4 and L_5 represent paths that have $hd(tl)$ labels that are not balanced by any $hd^{-1}(tl^{-1})$ labels. L_4 paths are "increasing" and correspond to execution paths in which the value at the source of the path is a proper substructure of the value at the target, while L_5 paths are "equal or increasing":

$$
\begin{aligned}
L_4\text{:}\quad & \mathit{incr\_path} \rightarrow \mathit{eq\_path}\ \ \mathbf{hd}\ \ \mathit{incr\_path} \\
& \mathit{incr\_path} \rightarrow \mathit{eq\_path}\ \ \mathbf{tl}\ \ \mathit{incr\_path} \\
& \mathit{incr\_path} \rightarrow \mathit{eq\_path}\ \ \mathbf{hd}\ \ \mathit{eq\_path} \\
& \mathit{incr\_path} \rightarrow \mathit{eq\_path}\ \ \mathbf{tl}\ \ \mathit{eq\_path}
\end{aligned}
$$

$$
\begin{aligned}
L_5\text{:}\quad & \mathit{eq\_or\_incr\_path} \rightarrow \mathit{eq\_path}\ \ \mathbf{hd}\ \ \mathit{eq\_or\_incr\_path} \\
& \mathit{eq\_or\_incr\_path} \rightarrow \mathit{eq\_path}\ \ \mathbf{tl}\ \ \mathit{eq\_or\_incr\_path} \\
& \mathit{eq\_or\_incr\_path} \rightarrow \mathit{eq\_path}
\end{aligned}
$$

- As illustrated by Example 5 below, an arbitrary path of flow edges from vertex u to vertex v may represent a "false" dependence because the values at u and v have no common substructure. The language of L_6 consists of all paths of flow edges that do not represent "false" dependences. In other words, if vertices u and v are connected by an L_6-path, the values at u and v have a common substructure along some execution path:

$$
L_6\text{:}\quad \mathit{flow\_path} \rightarrow \mathit{eq\_or\_decr\_path}\ \ \mathit{eq\_or\_incr\_path}
$$

Example 5 In the ADDG for program $\mathbf{P_1}$ shown in Example 4, the path from vertex *ops* to vertex *vals* with label $hd^{-1}.hd.id.tl^{-1}.id.id.id$ suggests that the value of *vals* depends on the value of *ops*. However, the path represents a "false" dependence, because the tl^{-1} (or *cdr*) operation on parameter *state* extracts its tail, whereas the *hd* (or *cons*) operation on *car*(*ops*) places *car*(*ops*) in the the head of *state*. Similarly, the path from *ops* to *vals* with label $tl^{-1}.id.hd^{-1}.hd.id.tl^{-1}.id.id.id$ represents a "false" dependence. □

The context-free language L_6 of all flow paths that do not represent "false" dependences can be split into two disjoint context-free languages that represent either decreasing behaviour or possibly increasing behaviour, as follows:

$$
L_7\text{:}\quad \mathit{insitu\_decr\_path} \rightarrow \mathit{decr\_path}
$$

The language of L_7 represents paths that are size-decreasing. We would like to test the in-situ-decreasing property for a parameter by testing whether every path from the parameter vertex to itself in the ADDG is in the language L_7. However, CFG-reachability can only be used to test whether there exists an L-path from a source to a target. Hence we need to define the language of all flow paths without "false" dependences that are not L_7-paths. This can be done as follows:

$$\begin{array}{lll} L_8: & insitu\_eq\_or\_incr\_path & \rightarrow eq\_or\_decr\_path \;\; incr\_path \\ & insitu\_eq\_or\_incr\_path & \rightarrow eq\_path^+ \\ & eq\_path^+ & \rightarrow \mathbf{hd} \;\; eq\_path^+ \;\; \mathbf{hd^{-1}} \;\; eq\_path^+ \\ & eq\_path^+ & \rightarrow \mathbf{tl} \;\; eq\_path^+ \;\; \mathbf{tl^{-1}} \;\; eq\_path^+ \\ & eq\_path^+ & \rightarrow \mathbf{id} \;\; eq\_path^+ \\ & eq\_path^+ & \rightarrow \mathbf{id} \end{array}$$

The paths in the language of L_8 are either "equal" or contain at least one $hd(tl)$ label that is not balanced by $hd^{-1}(tl^{-1})$. (Language *eq_path*$^+$ is like *eq_path*, but does not contain ε.) Such paths are not size-decreasing because the value at the target of the path may not be a substructure of the value at the source of the path. It can be shown that the languages L_7 and L_8 are disjoint and that their union is $L_6 - \varepsilon$. In other words, every L_6-path is either an L_7-path or an L_8-path, and no path is both an L_7-path and an L_8-path. Thus, verifying that no flow path from vertex u to vertex v is in *insitu_eq_or_incr_path* is sufficient to ensure that every flow path from u to v is either in *insitu_decr_path* or represents a "false" dependence.

The final path language of interest in the ADDG is the language of control paths. A control path from function f to function g indicates that a call on f may produce a call on g, and is a sequence of zero or more edges labeled with *control*:

$$\begin{array}{lll} L_9: & control\_path & \rightarrow \mathbf{control} \;\; control\_path \\ & control\_path & \rightarrow \varepsilon \end{array}$$

The languages of path labels defined in this section are all context free languages. Every source-target L-path problem for each of these languages can therefore be solved in time cubic in the number of nodes in the ADDG.

6 Linking ADDG paths and in-situ-decreasing behaviour

In this section we relate the presence or absence of various L-paths in the ADDG (for various languages L defined in the previous section) with the semantic characterization of in-situ-decreasing behaviour from Section 3. This serves as a formal justification of the algorithm for identifying in-situ-decreasing-parameters that is presented in Section 7.

We first relate the semantic concept of a transition sequence defined in Section 2 with the syntactic concept of a *call path* defined by Glensrup and Jones in [3]. A call path is an abstraction of a transition sequence where every vector of values is replaced by a vector of syntactic expressions. We show that every transition sequence from function f to itself corresponds to a call path from f to itself. We then relate the in-situ-decreasing property for parameters as expressed in terms of call paths with the presence or absence of certain paths in the ADDG.

6.1 Call paths

A *call path* of length $k-1$ is a sequence $\pi = [(f^1, \vec{x}), (f^2, \vec{e}^2), \ldots, (f^k, \vec{e}^k)]$, where p (assumed to be tail recursive) contains definitions $f^{i-1}\vec{y} = \ldots f^i\vec{e}^i \ldots$ for $2 \le i \le k$. $\vec{e}^2$ is a vector of arguments for f^2 obtained by unfolding the call from f^1 to f^2 without doing any computation, and is expressed in terms of $\vec{x}$. Similarly, $\vec{e}^k$ is obtained by unfolding the calls from f^1 to f^2 ... f^{k-1} to f^k without doing any computation, and is expressed as a function of $\vec{x}$. Argument $\vec{e}^k_j$ is said to "depend" on argument $\vec{x}_j$ iff the expression for $\vec{e}^k_j$ in terms of $\vec{x}$ contains the symbol $\vec{x}_j$. We define the size operator $\ll$ on expressions as follows: given two expressions e_1 and e_2, $e_1 \ll e_2$ iff $\forall \vec{v} : [\![\vec{e}_1]\!]\vec{v} < [\![\vec{e}_2]\!]\vec{v}$.

Example 6 In program $\mathbf{P_1}$ from Example 1, $[(eval, [o, v, t, e]), (eval, [cdr(o), cdr(v), t + car(v), e])]$ is a call path that represents a possible recursive call from *eval* to itself. □

Call paths are abstractions of transition sequences in which every vector of values is replaced by a vector of syntactic expressions. In general, every real computation or transition sequence is represented by some call path, although the reverse does not hold.

We can express the in-situ-decreasing property for function parameters in terms of call paths instead of transition sequences: If the expression $\vec{e}^k_j$ is strictly smaller in size than the expression $\vec{x}_j$ along some call path from f to f, the parameter f_j must strictly decrease in size along every transition sequence from f to itself that corresponds to the given call path. Hence if $\vec{e}^k_j$ is strictly smaller than $\vec{x}_j$ along all call paths from f to f, the parameter f_j must be in-situ-decreasing.

Lemma 1 A parameter f_j of function f in p is in-situ-decreasing if for every call path $\pi = [(f^1, \vec{x}), \ldots, (f^k, \vec{e}^k)]$ where $f^1 = f^k = f$, $\vec{e}^k_j \ll \vec{x}_j$. □

The re-statement of the in-situ-decreasing property through Lemma 1 has the advantage that call paths are directly related to paths in the ADDG. This is because the syntactic "dependence" between arguments f_i and g_j in a call path is directly represented in the ADDG by a path of flow edges from parameter vertex f_i to parameter vertex g_j.

6.2 Relating call paths and paths in the ADDG

In this subsection, we relate the in-situ-decreasing property for function parameters with the presence or absence of certain paths in the ADDG. This justifies the use of the CFG-reachability algorithm for identifying in-situ-decreasing parameters.

The observation below formalises the idea that in a recursive call from function f to itself, parameter f_j can depend on f_j itself, on other sibling parameters of f, on constant values, or on a combination of these.

Observation 1 In every call path $\pi = [(f^1, \vec{x}), \ldots, (f^k, \vec{e}^k)]$ in p where $f^1 = f^k = f$, $\vec{e}^k$ depends only on $\vec{x}$. □

This result is useful in justifying the lemmas that follow in this section, which link the existence of paths in ADDG G with semantic properties of parameters in program p.

An in-situ-decreasing parameter must satisfy two basic conditions: it must depend on itself in a size-decreasing manner, and it must not depend on sibling parameters or constants.

The presence of a flow-path from vertex f_j to itself ensures that parameter f_j depends on itself, while the lemma below says that if a parameter depends on itself and that dependence is one that is not-size-decreasing, there must be a path from f_j to f_j in the ADDG such that the path label is in the language *insitu_eq_or_incr_path*.

Lemma 2 If parameter f_j of function f in p is not in-situ-decreasing and there exists a call path $\pi = [(f^1, \vec{x}), \ldots, (f^k, \vec{e}^k)]$ in p where $f^1 = f^k = f$ and $\vec{e}^k_j$ depends only on $\vec{x}_j$, then there exists a path s from f_j to f_j in G such that s is an L_8-path (the concatenated label on s is in *insitu_eq_or_incr_path*.) □

Read as its contrapositive, Lemma 2 says that if all paths from f_j to f_j in G are size-decreasing, there can be no recursive call from f to itself where f_j depends only on itself and does not become strictly smaller in size. Thus f_j can only depend on itself in a size-decreasing manner.

If parameter f_j depends on a sibling parameter f_i, there must be a flow path from vertex f_i to vertex f_j in G. This is expressed in lemma 3 below.

Lemma 3 If there exists a call path $\pi = [(f^1, \vec{x}), \ldots, (f^k, \vec{e}^k)]$ in p where $f^1 = f^k = f$ and $\vec{e}^k_j$ depends on $\vec{x}_i$ where $i \neq j$, then there exists a path s from f_i to f_j in G such that s is an L_6-path (the concatenated label on s is in *flow_path*.) □

Thus, if there is no flow-path in G from a sibling parameter f_i to f_j, then the value of f_j can depend only on f_j itself or on constant values.

If parameter f_j takes on a constant value in a call from f to itself, there must be a flow-path from the associated constant vertex within the control structure of f to vertex f_j, as shown in the lemma below.

Lemma 4 If there exists a call path $\pi = [(f^1, \vec{x}), \ldots, (f^k, \vec{e}^k)]$ in p where $f^1 = f^k = f$ and $\vec{e}^k_j$ does not depend on $\vec{x}$, then there exists a constant vertex c in G such that there is an L_6-path from c to f_j in G and there is an L_9-path (with a concatenated label in *control_path*) from f to g in G where g is the function that encloses c. □

Thus, if there are no flow-paths to f_j from constant vertices within a call path from f to f, then f_j cannot take on constant values in any recursive calls on f.

To summarize: Given a parameter vertex f_j in G, the presence of an L_7-path (*insitu_decr_path*) from f_j to f_j along with the absence of any L_8-paths (*insitu_eq_or_incr_path*) from f_j to f_j guarantees that f_j depends on itself in a size-decreasing manner, while the absence of L_6-paths (*flow_paths*) from siblings f_i to f_j and from constant vertices c to f_j ensures that f_j depends only on itself. Hence under these conditions f_j must be in-situ-decreasing. This is formalized in Theorem 2 below.

Theorem 2 Parameter f_j of function f in p is in-situ-decreasing if all of the following properties hold:

(a) there exists an L_7-path (*insitu_decr_path*) from vertex f_j to vertex f_j in G, and there does not exist an L_8-path (*insitu_eq_or_incr_path*) from vertex f_j to vertex f_j in G, and
(b) there does not exist an L_6-path (*flow_path*) from vertex f_i to vertex f_j in G such that f_i is a sibling parameter of f_j, and
(c) there does not exist an L_6-path (*flow_path*) from constant vertex c to f_j in G such that there exists an L_9-path (*control_path*) from f to g enclosing c. □

Property (a) ensures that successive values of f_j in a recursive call on f are related to previous values of f_j in a size-decreasing manner, while properties (b) and (c) ensure that successive values of f_j depend *only* on the previous values of f_j.

In the terminology of Section 6.1, the three properties of Theorem 2 combine to provide a safe test for the condition $\vec{e}^k_j \ll \vec{x}_j$ on all call paths from f to f.

7 A safe algorithm for marking BSV parameters

In the previous section, we established the link between the presence or absence of certain kinds of paths in the ADDG and the in-situ-decreasing behaviour of function parameters. All the languages of path labels defined earlier are context-free languages. Hence, we can use CFL-reachability to determine whether any of these paths exists between any two vertices in the ADDG.

This section contains descriptions of three algorithms: an algorithm that uses CFL-reachability to identify a subset of all in-situ-decreasing parameters in a program, an algorithm that identifies a subset of all influential parameters, and an algorithm that uses the results of these two algorithms to identify a subset of all BSV parameters in a program.

7.1 Identifying in-situ-decreasing parameters

The algorithm to identify in-situ-decreasing vertices that is described here is a restatement of Theorem 2 from the previous section. A parameter f_j of function f is in-situ-decreasing if it depends on itself in a size-decreasing manner, it does not depend on any sibling vertex, and it does not take any constant value during a (recursive) chain of calls from f to f. Any parameter of a non-recursive function is also in-situ-decreasing.

Algorithm 1 (Identify in-situ-decreasing vertices) Mark as *isd* all parameter vertices that are members of the relation *isd* defined below:[2]

isd(v)	**if**	$\neg$*incr(v,v)* **and** *decr(v,v)* **and** $\neg$*const(v)* **and** $\neg$*sibl(v)*
isd(v)	**if**	$\neg$*rec(function(v))*
where		
incr(m,n)	**if**	*insitu_eq_or_incr_path(m,n)*
and		
decr(m,n)	**if**	*insitu_decr_path(m,n)*
and		
const(m)	**if**	*control_path(function(m),function(c))* **and** *flow_path(c,m)* **and** c is a constant vertex
and		
sibl(m)	**if**	*flow_path(n,m)* **and** *function(n) = function(m)* **and** n is a parameter vertex
and		
rec(f)	**if**	*control_path(f,f)* □

Example 7 The result of applying Algorithm 1 to program $\mathbf{P_1}$ from Example 1 is as follows: all parameters of functions *checkValid* and *accum* are marked *isd* because these functions are non-recursive (there is no *control_path* from *checkValid* to itself or from *accum* to itself.) Parameter *err* is not marked *isd* because there is an *insitu_eq_or_incr_path* from *err* to itself (with label *id*.) Similarly, *tot* is not marked *isd* because of the path with label *id.id.id.id* from *tot* to itself. Parameter *ops* is marked *isd* because the only paths to vertex *ops* are in *insitu_decr_path*. Finally, parameter *vals* of *eval* is marked *isd* because the only paths in *flow_path* that have *vals* as their target are in *insitu_decr_path*. □

Readers familiar with the work of Glenstrup and Jones [3] will notice that Algorithm 1 has a substantially different structure than their algorithm (quite apart from the use of CFL-reachability). In the algorithm of Glenstrup and Jones, parameters may be identified as in-situ-decreasing on the basis of the BSV behaviour of other parameters. At the same time, parameters may be identified as BSV if other vertices are in-situ-decreasing. This mutual dependence means that the two phases of their algorithm must be applied iteratively to identify the maximum number of BSV parameters.

In contrast, Algorithm 1 does not depend on information about the BSV behaviour of parameters in the program. Hence a single application of Algorithm 1 followed by Algorithm 3 (our algorithm that identifies BSV parameters) is sufficient to mark all BSV vertices that can be identified by our approach.

7.2 Identifying influential parameters

As pointed out in the introduction, an in-situ-decreasing parameter may not control the recursion of its associated function. If specialization continues after the parameter has taken on the value *null*, operations such as cdr(*null*) or car(*null*) may be performed. If the specializer does not detect such errors, infinite unrolling may result. Even if the specializer checks for such invalid operations, it cannot determine whether the error occured because of problematic code or because of "too much" unrolling.

In Section 3, we defined the influential property for function parameters: A parameter f_j of function f is influential if a *null* value for f_j guarantees that no further recursive calls to f take place. Because precise identification of the set of all parameters that satisfy this definition is not possible, we identify a subset of all parameters that are guaranteed to generate no further function calls (recursive or otherwise) when their value is *null*.

Algorithm 2 (Mark influential vertices) Given a parameter f_j of function f, the expression body of f is modified by replacing all occurences of f_j with *null* and by replacing all function calls with $\perp$. The

[2] We use the notation *foo_path(m,n)* to mean that that n is reachable from m via a path in the language *foo_path*.

expression is then simplified using a standard intra-procedural constant propagation technique such as the algorithm of Wegman and Zadeck in [14]. If the simplified expression does not contain any occurences of $\perp$, parameter f_j satisfies the influential property and f_j is marked as *influ*. This procedure is applied to every parameter in p. □

Example 8 The result of applying Algorithm 2 to program $\mathbf{P_1}$ from Example 1 is as follows: All parameters of functions *checkValid* and *accum* are marked as *influ* because these functions do not have any function calls in their expressions. Parameter *vals* of *eval* is marked as *influ* because the result of simplifying the expression body of *eval* with *vals* = *null* is the expression *err*, which is free of function calls. No other parameter of *eval* is marked as *influ*. □

7.3 Identifying BSV parameters

Algorithms 1 and 2 mark some function parameters as *isd* (in-situ-decreasing) and *influ* (influential). These markings can be used to identify a subset of all the BSV parameters in a program, as in Algorithm 3 below.

Algorithm 3 (Mark BSV vertices) The algorithm has two phases: In the first phase, a subset of all quasi-BSV parameters in a program are marked *qbsv*. Vertices are marked as *qbsv* on the basis of Definition 4 from Section 3. Parameters that are marked both *isd* and *influ* are identified as controlling and marked *cont*, following which the parameters marked *cont* and their siblings are marked *qbsv* (quasi-bounded-static-varying).
In the second phase of Algorithm 3, BSV markings from the static inputs of the main function are propagated through the ADDG according to the following rules: Initially, every known input parameter to the main function is marked *bsv*, while every other input parameter is marked D. At every non-parameter vertex, the vertex is marked *bsv* if all its predecessors are marked *bsv*. At every parameter vertex marked *qbsv*, the vertex is marked *bsv* if all its non-recursive predecessors (see below) are marked *bsv*. Every vertex is marked D if any of its predecessors is marked D. When no further vertices can be marked *bsv*, all vertices not marked *bsv* are marked D. □

A non-recursive predecessor of a parameter vertex f_j of function f is a predecessor of f_j that may define an entry value for f_j. Such a predecessor vertex must be enclosed by a function that cannot be called by f:

Definition 7 A vertex v in the ADDG G of program p is a *non-recursive predecessor* of parameter f_j of function f in p iff there is no L_9-path (*control_path*) from f to g where g is the function that encloses v. □

The marking rule used at parameter vertices marked *qbsv* in the second phase of the algorithm follows from Theorem 1 in Section 3. A parameter is initially-bounded if all of its non-recursive predecessors are BSV. If, in addition, the parameter is quasi-BSV, it must be BSV.

In Algorithm 3, steps 3 and 4 of Phase I and steps 3 and 4 of Phase II are all operations that are linear in the number of vertices in the ADDG, $(P + Var + Ops)$, where P is the number of parameters in p, Var is the number of *where* variables in p, and Ops is the number of cons/car/cdr operations in p.

8 Related work

The basis for the work described in this paper is Holst's definition of the in-situ-decreasing property for function parameters in [4]: An in-situ decreasing parameter of a function f strictly decreases in size on every (recursive) chain of calls from f to f.

Glenstrup and Jones define a second algorithm for identifying in-situ-decreasing parameters, which uses the markings $\uparrow$, $\downarrow$ and $=$ on edges in the parameter dependency graph [3]. The algorithm described in this paper extends their work by using more precise markings on flow edges and by using CFL-reachability to identify a broader class of size-decreasing paths.

Andersen and Holst have described an extension of Holst's analysis to a higher-order lambda calculus [2]. Although their primary emphasis was termination analysis for programs with higher-order functions, they observe that their technique for handling higher-order functions can be adapted to discover some size-decreasing paths containing $\uparrow$ edges.

Our approach was conceived independently of their work, but we became aware of it shortly after it was presented at SAS this fall. Some of the differences between their work and our approach are:

- Their approach is based on tree grammars, whereas our approach is an extension of Glenstrup and Jones's approach, which is based on graph reachability. In order to identify a greater number of in-situ-decreasing parameters than Glenstrup and Jones, we extend the parameter dependency graph with new nodes and new edge markings and we use CFL-reachability rather than a closed semi-ring graph algorithm [1, 3].
- There is a general result that all "context-free language reachability problems" can be solved in time cubic in the number of vertices in the graph [15]. This allows us to obtain a cubic-time bound for our algorithm. (Andersen and Holst did not report a bound on the running time of their algorithm, although we suspect that for the class of problems we are addressing, their methods would also run in cubic time.)
- We are able to provide a semantic justification for our method by showing that every parameter identified as in-situ-decreasing by our algorithm is semantically in-situ-decreasing.

CFL-reachability has also been used for a number of other program-analysis problems: Reps, Sagiv, and Horwitz applied CFL-reachability techniques to interprocedural dataflow-analysis problems [11, 12] and to inter-procedural slicing [10]. Reps used CFL-reachability to develop a shape-analysis algorithm [9]. He used edge markings that are identical to the markings used on the edges of the ADDG defined in this paper.

Melski and Reps have shown that CFL-reachability problems are convertible into a class of set-constraint problems (and vice versa) [8]. Because set-constraints are related to regular-tree grammars, this result also has some bearing on the relationship between our work and that of Andersen and Holst. The exact relationship issomewhat fuzzy at this point because the tree grammars used in the Andersen and Holst paper do not make a direct application of the Melski-Reps result possible. (Even if it were, the details of the general-case construction would obscure the relatively simple concepts expressed by the context-free grammars given in Section 5.)

References

[1] A.V. Aho, J.E. Hopcroft, and J.D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading, MA, 1974.

[2] P. H. Andersen and C. K. Holst. Termination analysis for offline partial evaluation of a higher order functional language. In *Proceedings of the Third International Static Analysis Symposium*, 1996.

[3] Arne J. Glenstrup and Neil D. Jones. BTA algorithms to ensure termination of off-line partial evaluation. *Andrei Ershov Second International Conference 'Perspectives of System Informatics', Lecture Notes in Computer Science, 1996.*

[4] C.K. Holst. Finiteness analysis. In J. Hughes, editor, *Functional Programming Languages and Computer Architecture, Cambridge, Massachusetts, August 1991 (Lecture Notes in Computer Science, vol. 523)*, pages 473–495. ACM, Berlin: Springer-Verlag, 1991.

[5] N.D. Jones. Automatic program specialization: A re-examination from basic principles. In D. Bjørner, A.P. Ershov, and N.D. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 225–282. Amsterdam: North-Holland, 1988.

[6] N.D. Jones, C.K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation.* Englewood Cliffs, NJ: Prentice Hall, 1993.

[7] J. Kasami. An efficient recognition and syntax analysis algorithm for context-free languages. Scientific Report AFCRL-65-758, Air Force Cambridge Research Laboratory, Bedford, MA, 1965.

[8] D. Melski and T. Reps. Interconvertibility of set constraints and context-free langauge reachability. Technical Report 1330, Computer Sciences Department, University of Wisconsin-Madison, November 1996.

[9] T. Reps. Shape analysis as a generalized path problem. In *Partial Evaluation and Semantics-Based Program Manipulation, La Jolla, California, June 1995*, pages 1–11. New York: ACM, 1995.

[10] T. Reps, S. Horwitz, M. Sagiv, and G. Rosay. Speeding up slicing. In *SIGSOFT 94: Proceedings of the Second ACM SIGSOFT Symposium on the Foundations of Software Engineering, (New Orleans, LA, December 7-9, 1994), ACM SIGSOFT Software Engineering Notes 19(5)*, pages 11–20, December 1994.

[11] T. Reps, M. Sagiv, and S. Horwitz. Interprocedural dataflow analysis via graph reachability. Technical Report 94/14, DIKU, University of Copenhagen, Denmark, April 1994.

[12] T. Reps, M. Sagiv, and S. Horwitz. Precise interprocedural dataflow analysis via graph reachability. In *Conference Record of the Twenty-Second ACM Syposium on Principles of Programming Languages, (San Francisco, CA, Jan. 23-25, 1995)*, pages 49–61, 1995.

[13] P. Sestoft. Automatic call unfolding in a partial evaluator. In D. Bjørner, A.P. Ershov, and N.D. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 485–506. Amsterdam: North-Holland, 1988.

[14] M. Wegman and K. Zadeck. Constant propagation with conditional branches. In *Conference Record of the Twelfth ACM Syposium on Principles of Programming Languages*, pages 291–299, 1985.

[15] M. Yannakakis. Graph-theoretic methods in database theory. In *Proceedings of the Symposium on Principles of Database Systems, 1990*, pages 230–242, 1990.

[16] D.H. Younger. Recognition and parsing of context-free languages in time n^3. *Information and Control*, (10):189–208, 1967.